User input/output, error handling and modules Part I

Foundation of programming (CK0030)

Francesco Corona

FdP

- ☺ Intro to variables, objects, modules, and text formatting
- ☺ Programming with WHILE- and FOR-loops, and lists
- ☺ Functions and IF-ELSE tests

- ☹ **Data reading and writing**
- ☹ Error handling
- ☹ Making modules

- ☹ Arrays and array computing
- ☹ Plotting curves and surfaces

FdP (cont.)

Example

Consider a program to calculate the sine formula $x = \sin(\omega t)$

```
1 from math import sin
2
3 A = 0.1
4 w = 1
5 t = 0.6
6
7 x = A*sin(w*t)
8
9 print x
```

`A`, `w` and `t` are parameters that must be known before the program starts performing the calculation of `x`

- Parameters `A`, `w` and `t` are thought as input data
- Consistently, `x` constitute the output data

FdP (cont.)

Here, input data are hardcoded within in the program

- Corresponding variables are explicitly set to specific values
- `A=0.1`, `w=1`, `t=0.6`

```
1 from math import sin
2
3 A = 0.1
4 w = 1
5 t = 0.6
6
7 x = A*sin(w*t)
8
9 print x
```

This programming style may be suitable for small programs

FdP (cont.)

In general, it is good practice to let the program user provide input data

- No need to modify the program itself for new sets of input data

We discuss three different ways of reading data into a program:

1. Let the user answer questions in a dialog in the terminal window
2. Let the user provide input on the command line
3. Let the user provide input data in a file

FdP (cont.)

Even if the program is working perfectly, wrong input data from the user may cause the program to produce wrong answers or even crash

- Checking that input data are correct is important
- We discuss how to this using exceptions

FdP (cont.)

The Python environment is organised as a large collection of modules

- It is important to know how to organise your own software
- We discuss how to do this using modules themselves

Questions and answers
User input, error handling and modules

Questions and answers

Perhaps the simplest way of getting data into a program is trivial

1. Ask the user a question
2. Let the user type in an answer
3. Read the text in the answer into a variable in the program

This is done by calling `function raw_input` in Python 2.x

- Or, equivalently, `function input` in Python 3.x

Keyboard input
Questions and answers

Keyboard input

The temperature conversion from Celsius to Fahrenheit: $F = \frac{9}{5}C + 32$

The conventional program with variable `C` set explicitly in the program

```
1 C = 22
2 F = 9./5*C + 32
3 print F
```

Example

We ask the question '`C=?`' and wait for the user to input a number

- The program can read the number and store it in a variable `C`

```
1 C = raw_input('C=? ')
```

- Python, then, waits until the user presses the `Return` key

The `raw_input function` returns the input as a `string object`

- Variable thus `C` refers to a `string object`

Keyboard input (cont.)

Before use, the string must be first converted to floating-point number

- `C = float(C)`

```
1 C = raw_input('C=? ')
2 C = float(C)
3
4 F = 9.0/5*C + 32
5 print F
```

The execution of the program and the resulting dialog with the user

```
1 Terminal > python c2f_qa.py
2
3 C=? 21
4 69.8
```

The `raw_input function` reads the characters `21` from the keyboard

- It returns the string `21`, which we refer to as variable `C`

Reading from command line

User input and error handling

Reading from command line

Programs often avoid asking questions, data are rather fetched from CL

- How can we access information on the command line in Python?

Input from command line

Reading from command line

Input from command line (cont.)

Example

Consider the Celsius-Fahrenheit conversion program, with the idea of providing the Celsius input temperature as a command-line argument

- We want a program that to be executed requires the input temperature as input

```
1 Terminal > python c2f_cml.py 21
2
3 69.8
```

Input from command line (cont.)

Inside the program, text `21` is fetched as `sys.argv[1]`

```
1 Terminal > python c2f_cml.py 21
2
3 69.8
```

Definition

`Module sys` has a list `argv` containing all command-line arguments

- All 'words' appearing after the program name
- Here, one argument, stored in `sys.argv[1]`

Remark

The first element in `sys.argv`, `sys.argv[0]`, is the program name

Input from command line (cont.)

As with keyboard inputs, command-line arguments are treated as text

- `sys.argv[1]` refers to `string object` whose value here is `21`

Example

```
1 import sys
2
3 C = float(sys.argv[1])
4 F = 9.0*C/5 + 32
5
6 print F
```

As we interpret the command-line argument as a number and we want to compute with it, it is necessary to convert the string to a float object

Input from command line (cont.)

Example

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

```
1 v0 = 5
2 g = 9.81
3 t = 0.6
4 y = v0*t - 0.5*g*t**2
5
6 print y
```

Instead of hard-coding `v0` and `t`, they can be read from command line

```
1 Terminal > python ball2_cml.py 0.6 5
2
3 1.2342
```

Input from command line (cont.)

Command-line arguments are given as `sys.argv[1]` and `sys.argv[2]`

```
1 import sys
2
3 t = float(sys.argv[1])
4 v0 = float(sys.argv[2])
5
6 g = 9.81
7
8 y = v0*t - 0.5*g*t**2
9 print y
```

Command line arguments
Reading from command line

Command line arguments

Example

Let us code `addall.py` that adds all its command-line arguments

```
1 Terminal > python addall.py 1 3 5 -9.9
2
3 The sum of 1 3 5 -9.9 is -0.9
```

Command line arguments

```
Terminal > python addall.py 1 3 5 -9.9

The sum of 1 3 5 -9.9 is -0.9
```

A first possible solution

```
import sys

s = 0
for arg in sys.argv[1:]:
 number = float(arg)      # Need to convert to float to compute
 s += number

print 'The sum of ',
for arg in sys.argv[1:]:
 print arg,               # No need to convert to float to print
print 'is ', s
```

The command-line arguments are stored in sublist `sys.argv[1:]`

- Each element is a string, needs conversion to float

The output is one line, built from print statements with trailing comma

Command line arguments (cont.)

```
1 Terminal > python addall.py 1 3 5 -9.9
2
3 The sum of 1 3 5 -9.9 is -0.9
```

An alternative, compact solution

```
1 import sys
2
3 s = sum([float(x) for x in sys.argv[1:]])
4
5 print 'The sum of %s is %s' % (' '.join(sys.argv[1:]), s)
```

String list `sys.argv[1:]` is first converted to a list of float objects

- Then, it is passed to `sum` for adding the numbers

Command line arguments (cont.)

Definition

Construction `S.join(L)` places the elements in list `L` after each other

- with string `S` in between

The result is a string with all the elements in `sys.argv[1:]` and a space in between, which is the text shown as command line output

Command line arguments (cont.)

Unix commands make heavy use of command-line arguments

Example

- Typing `ls -s -t` lists files in the current folder, and program `ls` is run with two command-line arguments, `-s` and `-t`
- `-s` tells `ls` to print the file name together with the file size
- `-t` sorts the list according to dates of last modification

Example

- Typing `cp -r my new` copies folder-tree `my` to a new folder-tree `new` by invoking program `cp` program with three command-line arguments: `-r` (for recursive copying of files), `my`, and `new`

Command line arguments (cont.)

Most programming languages have support for extracting the CL args

- An important rule is that CL arguments are separated by blanks

What if we want to provide a text containing blanks as CL args?

- The text containing blanks must then appear inside single or double quotes

Command line arguments (cont.)

Example

We demonstrate this with a program that simply prints CL args

```
1 import sys, pprint
2
3 pprint.pprint(sys.argv[1:])
```

```
1 Terminal > python print_cml.py 21 a string with blanks 31.3
2
3 ['21', 'a', 'string', 'with', 'blanks', '31.3']
```

```
1 Terminal > python print_cml.py 21 "a string with blanks" 31.3
2
3 ['21', 'a string with blanks', '31.3']
```

User text into objects
User input and error handling

User text into objects

It is possible to provide text with valid code as input to a program, and turn it into objects as if the text were written directly into the program

- It is a powerful tool for letting users input expressions

The code has no knowledge of the kind of function the user is inputting

The EVAL function

User text into objects

The EVAL function

Function `eval` takes a string as argument and evaluates it as **expression**

- The result of an `expression` is an object

Example

```
>>> r = eval('1+2')
>>> r
    3

>>> type(r)
    <type 'int'>
```

`r = eval('1+2')` is equivalent to writing `r = 1+2` directly

```
>>> r = 1+2
>>> r
    3

>>> type(r)
    <type 'int'>
```

The EVAL function (cont.)

Remark

Valid Python `expressions` stored as text in string `s`
can be turned into live Python code using `eval(s)`

The EVAL function (cont.)

Example

The string to be evaluated is `2.5`, which is understood as `r = 2.5`

```
>>> r = eval('2.5')
>>> r
    2.5

>>> type(r)
<type 'float'>
```

Example

```
>>> r = eval('[1, 6, 7.5]')
>>> r
    [1, 6, 7.5]

>>> type(r)
    <type 'list'>
```

We can initialise a list inside quotes and use `eval` to create a `list object`, equivalently to writing the assignment `r = [1, 6, 7.5]`

The EVAL function (cont.)

Example

We can make a `tuple object` by using tuple syntax (parentheses)

```
1 >>> r = eval('(-1, 1)')
2 >>> r
3     (-1, 1)
4
5 >>> type(r)
6     <type 'tuple'>
```

The EVAL function (cont.)

Example

```
>>> from math import sqrt
>>> r = eval('sqrt(2)')
>>> r
    1.4142135623730951

>>> type(r)
    <type 'float'>
```

This syntax is valid as long as `function sqrt` has been defined

The EVAL function (cont.)

A quoted string inside an `expression string` gives a `string object`

Example

```
1 >>> r = eval('"math programming"')
2 >>> r
3     'math programming'
4
5 >>> type(r)
6     <type 'str'>
```

Note the use of two types of quotes:

- First double quotes mark `math programming` as `string object`
- The other set of quotes, here single quotes (but it could have been triple single quotes), to embed the text inside a string

The EVAL function (cont.)

Remark

Irrelevant if we have single or double quotes as inner or outer quotes

- '"..."' is the same as "'...'", ' and " are interchangeable as long as a pair of either type is used consistently

The EVAL function (cont.)

Example

The next two expression strings are equivalent, although invalid

```
>>> r = eval('math programming')
>>> r = math programming
```

math and programming are two variables separated by a space (uh?)

```
>>> r= 'math programming'    # initialise some string r
>>> s= "'math programming'" # eval(s) evaluates all text in ""
```

The EVAL function (cont.)

So, why is the `eval function` so useful?

Input via `raw_input` or `sys.argv` is returned as `string object`

- Often, it must be converted to another type (`int` or `float`)

Sometimes, it is preferable to avoid specifying one particular type

- The `eval function` can then be of help

We feed the `string object` from the input to `eval function`

- Let it interpret the string and convert it to the right object

The EVAL function (cont.)

Example

Consider a simple program in which we read in two values and add them

Values could be `string`, `float`, `int`, `list objects`, and so forth

- As long as we can apply the `+` operator to the values

Since we do not know whether the user will input an actual `string`, a `float`, an `integer` or anything else, we convert the input by `eval`

- The user's syntax will determine the type

```
1 i1 = eval(raw_input('Give input: '))
2 i2 = eval(raw_input('Give input: '))
3
4 r = i1 + i2
5
6 print '%s + %s is %s\nwith value %s' % \
7         (type(i1), type(i2), type(r), r)
```

The EVAL function (cont.)

```
Terminal > python add_input.py

Give input: 4
Give input: 3.1

<type 'int'> + <type 'float'> is
 <type 'float'> with value 7.1
```

```
Terminal > python add_input.py

Give input: [-1, 3.2]
Give input: [9,-2,0,0]

<type 'list'> + <type 'list'> is <type 'list'>
 with value [-1, 3.2000000000000002, 9, -2, 0, 0]
```

The EVAL function (cont.)

```
Terminal > python add_input.py

Give input: 'one string'

Give input: " and another string"

<type 'str'> + <type 'str'> is <type 'str'>
 with value one string and another string
```

```
Terminal > python add_input.py

Give input: 3.2
Give input: [-1,10]

Traceback (most recent call last): File "add_input.py",
 line 3, in <module> r = i1 + i2

TypeError: unsupported operand type(s) for +:
 'float' and 'list'
```

The EVAL function (cont.)

This program is similar, except that it requires two CL input arguments

Example

```
1 import sys
2
3 i1 = eval(sys.argv[1])
4 i2 = eval(sys.argv[2])
5
6 r = i1 + i2
7 print '%s + %s becomes %s\nwith value %s' % \
8         (type(i1), type(i2), type(r), r)
```

The EVAL function (cont.)

An important example on the usefulness of `eval` is to turn formulas, given as input, into mathematics in the program

Example

```
1 from math import *          # make all math functions available
2 import sys
3
4 formula = sys.argv[1]
5 x = eval(sys.argv[2])
6
7 result = eval(formula)
8 print '%s for x=%g yields %g' % (formula, x, result)
```

- Say, passed formula is `2*sin(x) + 1` and passed number is `3.14`
- We get `formula = 2*sin(x) + 1` and `x = 3.14`

```
1 Terminal > python eval_formula.py "2*sin(x)+1" 3.14
2
3 2*sin(x)+1 for x=3.14 yields 1.00319
```

The EVAL function (cont.)

```
1 from math import *          # make all math functions available
2 import sys
3
4 formula = sys.argv[1]
5 x = eval(sys.argv[2])
6
7 result = eval(formula)
8 print '%s for x=%g yields %g' % (formula, x, result)
```

Passing `sin(x)` in the first CL argument requires to have `sin` defined

- All functions from `math module` are imported first

The EXEC function

User text into objects

The EXEC function

`Function exec` executes a string containing arbitrary code

- The string is not necessarily an `expression string`

The EXEC function (cont.)

Suppose the user provides a formula expression as input to the program

- The formula is available to us in the form of a `string object`
- We want to turn this formula into a callable Python `function`

Example

For formula `sin(x)*cos(3*x) + x**2`, we would write the `function`

```
1 def f(x):
2  return sin(x)*cos(3*x) + x**2
```

With `exec` we first construct the syntax for defining `f(x)` in a string

- Then, we apply `exec` to the `string object`

The EXEC function (cont.)

```
1 formula = sys.argv[1]
2
3 code = """
4 def f(x):
5  return %s
6 """ % formula
7 from math import *       # make sure we have sin, cos, exp, ...
8 exec(code)
```

Think of `sin(x)*cos(3*x) + x**2` as first command-line argument

- `formula` will hold this text, which is inserted into `code` string

`exec(code)` executes the code as if it had been written into the code

```
1 """
2 def f(x):
3  return sin(x)*cos(3*x) + x**2
4 """
```

The EXEC function (cont.)

Remark

`exec` can be used to turn user-provided formulas into Python `functions`

The EXEC function (cont.)

Example

We have some `function` for computing a definite integral $\int_a^b f(x)\mathrm{d}x$

- Say, using the midpoint rule over n intervals

```
1 def midpoint_integration(f, a, b, n=100):
2  h = (b - a)/float(n)
3  I = 0
4  for i in range(n):
5   I += f(a + i*h + 0.5*h)
6  return h*I
```

We want to read the interval limits `a` and `b`, `n` from command line

- as well as the formula `f` that defines $f(x)$

The EXEC function (cont.)

```
1  from math import *
2  import sys
3
4  f_formula = sys.argv[1]
5  a = eval(sys.argv[2])
6  b = eval(sys.argv[3])
7
8  if len(sys.argv) >= 5:
9   n = int(sys.argv[4])
10 else:
11  n = 200
```

- We import the entire `math module`
- We use `eval` when reading input for `a` and `b`
- This allows the user to provide values like `2*cos(pi/3)`

Next step is to convert the `f_formula` for $f(x)$ into a `function g(x)`

```
1  code = """
2  def g(x):
3   return %s
4  """ % f_formula
5
6  exec(code)
```

The EXEC function (cont.)

Given an ordinary `g(x)`, we ask the integration `function` to integrate

```
1 I = midpoint_integration(g, a, b, n)
2
3 print 'Integral of %s on [%g, %g] with n=%d: %g' % \
4       (f_formula, a, b, n, I)
```

The EXEC function (cont.)

```
def midpoint_integration(f, a, b, n=100):
 h = (b - a)/float(n)
 I = 0
 for i in range(n):
  I += f(a + i*h + 0.5*h)
 return h*I

from math import *
import sys

f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])

if len(sys.argv) >= 5:
 n = int(sys.argv[4])
else:
 n = 200

code = """
def g(x):
 return %s
""" % f_formula
exec(code)

I = midpoint_integration(g, a, b, n)
print 'Integral of %s on [%g, %g] with n=%d: %g' % \
      (f_formula, a, b, n, I)
```

The EXEC function (cont.)

A sample run with $\int_0^{\pi/2} \sin(x)dx$

```
1 Terminal > python integrate.py "sin(x)" 0 pi/2
2
3 Integral of sin(x) on [0, 1.5708] with n=200: 1
```

Strings into functions

User text into objects

Strings into functions

It is handy to get a user-provided formula and turn it into a `function`

- The book offers a tool for this task, `StringFunction`

Example

```
>>> from scitools.StringFunction import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula)   # Formula into function f(x)
```

```
>>> f(0)
    0.0

>>> f(pi)
    2.8338239229952166e-15

>>> f(log(1))
    0.0
```

Strings into functions (cont.)

Expressions involving more independent variables are also possible

Example

An example with function $g(t) = Ae^{-at}\sin(\omega x)$

```
1 g = StringFunction('A*exp(-a*t)*sin(omega*x)',
2                    independent_variable='t',
3                    A=1, a=0.1, omega=pi, x=0.5)
```

First argument is the formula then the independent variable's name

- 'x' is default

The other parameters (A, a, ω, and x) must be specified with values

```
1 g.set_parameters(omega=0.1)
2 g.set_parameters(omega=0.1, A=5, x=0)
```

Strings into functions (cont.)

Calling `g(t)` works as if `g` were a Python `function` of `t` that stores also parameters `A`, `a`, `omega`, and `x`, and their values

Remark

`pydoc` brings up more documentation on `StringFunction`

```
1 pydoc scitools.StringFunction.StringFunction
```

Remark

`StringFunction objects` are computationally as efficient as the usual user-defined `functions`

Option-value pairs
User input and error handling

Option-value pairs

CL arguments require the user to type all arguments in a sequence

- As when calling a `function` with `positional arguments`

How to assign CL arguments the same way as `keyword arguments`?

Arguments would need to be associated with a name, so that their sequence can be arbitrary, and only the arguments whose default value is not appropriate would need to be provided

- Such CL arguments may have `option-value` pairs
- With `option`/`value` is some name/valuepair for the argument

Option-value pairs (cont.)

Example

Consider the location $s(t)$ of an object at time t, given initial position $s = s_0$ at $t = t_0$ with initial velocity v_0, and constant acceleration a

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2$$

The formula requires four input variables

- `s0`, `v0`, `a`, and `t`

Program `location.py` takes four options

- `-s0`, `-v0`, `-a`, and `-t`

Option-value pairs (cont.)

```
1 Terminal > python location.py --t 3 --s0 1 --v0 1 --a 0.5
```

- The sequence of `option-value pairs` is arbitrary
- All options have a default value (no need to specify all options on the command line)
- All inputs should have sensible default values

Let $s_0 = 0$, $v_0 = 0$, $a = 1$ and $t = 1$ by default, and only change t

```
1 Terminal > python location.py --t 3
```

Example

```
# Set default values
s0 = v0 = 0; a = t = 1

import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--v0', '--initial_velocity', type=float,
                    default=0.0, help='initial velocity',
                    metavar='v')
parser.add_argument('--s0', '--initial_position', type=float,
                    default=0.0, help='initial position',
                    metavar='s')
parser.add_argument('--a', '--acceleration', type=float,
                    default=1.0, help='acceleration',
                    metavar='a')
parser.add_argument('--t', '--time', type=float, default=1.0,
                    help='time', metavar='t')
args = parser.parse_args()

s0 = args.s0; v0 = args.v0; a = args.a; t = args.t
s = s0 + v0*t + 0.5*a*t**2

print """
An object, starting at s=%g at t=0 with initial
velocity %s m/s, and subject to a constant
acceleration %g m/s**2, is found at the
location s=%g m after %s seconds.
""" % (s0, v0, a, s, t)
```

The ARGPARSE module
Option-value pairs

The ARGPARSE module

Python has `module argparse` for parsing `option-value` pairs

The use of `argparse` consists of three steps

The ARGPARSE module (cont.)

First, a parser object must be created

```
1 import argparse
2 parser = argparse.ArgumentParser()
```

The ARGPARSE module (cont.)

Second, the various command-line options are defined

```
1  parser.add_argument('--v0', '--initial_velocity', type=float,
2                      default=0.0, help='initial velocity',
3                      metavar='v')
4
5  parser.add_argument('--s0', '--initial_position', type=float,
6                      default=0.0, help='initial position',
7                      metavar='s')
8
9  parser.add_argument('--a', '--acceleration', type=float,
10                     default=1., help='acceleration',
11                     metavar='a')
12
13 parser.add_argument('--t', '--time', type=float, default=1.0,
14                     help='time', metavar='t')
```

First arguments to `parser.add_argument` is a set of names/options

- Names/options to be associated with an input parameter

Optional arguments are `type`, a `default` value, a `help` string, and a name (`metavar`) for the value of the argument in a usage string

The ARGPARSE module (cont.)

Remark

The `argparse module` automatically allows for an option `-h` or `-help` that prints a usage string for all the registered options

Remark

By default, `type` is `str`, the `default` value is `None`, the `help` string is empty, `metavar` is the option/name in upper case without dashes ('`-`')

The ARGPARSE module (cont.)

Third, we must read the command line arguments and interpret them

```
1 args = parser.parse_args()
```

Thru the `args` object, values of registered parameters are extracted

- `args.v0`
- `args.s0`
- `args.a`
- `args.t`

Remark

Parameter's name is set by the first option to `parser.add_argument`

The ARGPARSE module (cont.)

Example

```
parser.add_argument('--initial_velocity', '--v0', type=float,
                    default=0.0, help='initial velocity')
```

The initial velocity value will thus appear as `args.initial_velocity`

We can the add the `dest` keyword to specify where the value is stored

```
parser.add_argument('--initial_velocity', '--v0', dest='V0',
                    type=float, default=0.0,
                    help='initial velocity')
```

`args.V0` retrieves the value of the initial velocity

Remark

If no default value is provided, default is `None`

The ARGPARSE module (cont.)

Example

The example is completed by either evaluating s

```
1 s = args.s0 + args.v0*t + 0.5*args.a*args.t**2
```

or, by introducing new variables to match math ...

```
1 s0 = args.s0; v0 = args.v0; a = args.a; t = args.t
2 s = s0 + v0*t + 0.5*a*t**2
```

Math expressions as values
Option-value pairs

Math expressions as values

Values on command line involving mathematical symbols and functions, say `-v0 'pi/2'`, pose a problem with the code

The `argparse module` will try to do `float('pi/2')`

- `pi` is an undefined name

Changing `type=float` to `type=eval` is required to interpret `pi/2`

- `eval('pi/2')` fails too, as `pi` is not defined inside the `argparse module`

There are various remedies for this problem

Math expressions as values (cont.)

Write a function for converting a string value to the desired object

Example

```
1 def evalcmlarg(text):
2  return eval(text)
3
4 parser.add_argument('--s0', '--initial_position',
5                      type=evalcmlarg, default=0.0,
6                      help='initial position')
```

This way, `eval` is executed in the programmer's namespace, where (hopefully) `pi` or other symbols are imported

Explicit type conversion through a user-provided conversion function

```
"""
As location.py, with function to interpret CL strings via eval
"""
s0 = v0 = 0; a = t = 1                          # Set default values

from math import *
def evalcmlarg(text):
 return eval(text)

import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--v0', '--initial_velocity', default=0.0,
                    type=evalcmlarg, help='initial velocity')
parser.add_argument('--s0', '--initial_position', default=0.0,
                    type=evalcmlarg, help='initial position')
parser.add_argument('--a', '--acceleration', default=1.0,
                    type=evalcmlarg, help='acceleration')
parser.add_argument('--t', '--time', type=evalcmlarg,
                    default=1.0, help='time')
example = "--s0 'exp(-4.2)' --v0 pi/4"
args = parser.parse_args()

s0 = args.s0; v0 = args.v0; a = args.a; t = args.t
s = s0 + v0*t + 0.5*a*t**2

print """
An object, starting at s=%g at t=0 with initial velocity
%s m/s, and subject to a constant acceleration %g m/s**2,
is found at the location s=%g m after %s seconds.
""" % (s0, v0, a, s, t)
```

Math expressions as values (cont.)

More sophisticated conversions are possible

Example

Say s_0 is specified in terms of a function of some parameter p

$$s_0 = (1 - p^2)$$

String `-s0` can be used and `StringFunction` turns it into `function`

```
1 def toStringFunction4s0(text):
2  from scitools.std import StringFunction
3  return StringFunction(text, independent_variable='p')
4
5 parser.add_argument('--s0', '--initial_position',
6                      type=toStringFunction4s0, default='0.0',
7                      help='initial position')
```

A CL argument `-s0 'exp(-1.5) + 10(1-p**2)'` results in `args.s0` being a `StringFunction object`, to be evaluated for some `p` value

```
1 s0 = args.s0
2 p = 0.05
3 ...
4 s = s0(p) + v0*t + 0.5*a*t**2
```

Math expressions as values (cont.)

An alternative is to perform the correct conversion of values after the `parser object` has read the values

We treat argument types as strings in `parser.add_argument` calls

- We replace `type=float` by setting `type=str`
- (which is also the default choice of `type`)

Remark

The approach requires specification of default values as strings (say '0')

Math expressions as values (cont.)

Example

```
parser.add_argument('--s0', '--initial_position', type=str,
                    default='0', help='initial position')

...

from math import *
args.v0 = eval(args.v0)
# or
v0 = eval(args.v0)
s0 = StringFunction(args.s0, independent_variable='p')
p = 0.5

...

s = s0(p) + v0*t + 0.5*a*t**2
```

Data from file
User input, error handling and modules

Data from file

Data into a program from CL, or Q/A in terminal works for small data

- Otherwise, input data must be fetched from files

The task is to understand how programs are used read and write files

Remark

Before letting a program read the file, we must know the file format

- The structure of the text influences the set of statements needed to read the file

Data from file (cont.)

Example

Suppose we have recorded some measurement data in a file `data.txt`

- Read measurements in `data.txt`, find the average, print it

We start by viewing the content of file `data.txt`

- Load the file into a text editor or viewer

Unix/Mac: `emacs`, `vi/vimm`, `more/less`, `pico/nano`

Windows/DOS: `WordPad`, `NotePad++`, `type`

Data from file (cont.)

What we see is a column with numbers

```
21.8
18.1
19
23
26
17.8
```

Read the column of numbers into a list

- Compute their average

Reading line-by-line
Data from file

Reading line-by-line

Before the content of a file is read, the file must be first **opened**

```
1 infile = open('data.txt', 'r')
```

The action creates a `file object`, here stored in the variable `infile`

- Second argument to `open function`, string '`r`',
  tells that we want to open the file for reading

After the file is read, `file object` must be closed (`infile.close()`)

Reading line-by-line

Remark

A file can be opened for writing, by providing '`w`' as the second argument

Reading line-by-line

```
21.8
18.1
19
23
26
17.8
```

The basic technique for reading the file content is line-by-line

```
infile = open('data.txt', 'r')

for line in infile:
 # do something with line
```

The `line` variable is a `string object` holding the current line in file

- The `FOR-loop` that loops over the lines in the file has the same syntax as when we go through a list

Reading line-by-line

```
1 infile = open('data.txt', 'r')
2
3 for line in infile:
4  # do something with line
```

The file object infile can be understood as a collection of elements (lines in a file) and the FOR-loop visits these elements in a sequence

- At every pass, the line variable refers to one line

Remark

It is useful to do a print line inside the loop, if something goes wrong

Reading line-by-line

Instead of reading line-by-line, we can load all lines into a list of strings

```
1 infile = open('data.txt', 'r')
2
3 lines = infile.readlines()
```

Remark

The `lines = infile.readlines()` statement is equivalent to

```
1 lines = []
2 for line in infile:
3  lines.append(line)
```

or to the list comprehension

```
1 lines = [line for line in infile]
```

Reading line-by-line (cont.)

Example

Load the file into list `lines`, and compute the average of the numbers

A straightforward sum of all numbers on all lines, returns an error

```
1 infile = open('data.txt', 'r')
2
3 lines = infile.readlines()
4
5 mean=0
6 for number in lines:
7  mean = mean + number
8
9 mean = mean/len(lines)
```

Reading line-by-line (cont.)

```
1 TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

`lines` holds each line (number) as a string, not as `float` or `int`

```
1  infile = open('data.txt', 'r')
2
3  lines = infile.readlines()
4
5  mean = 0
6  for line in lines:
7   number = float(line)              # Convert each line to float
8   mean = mean + number
9
10 mean = mean/len(lines)
```

Reading line-by-line (cont.)

Example

A complete version of the code

```
1  infile = open('data.txt', 'r')
2
3  lines = []
4  for line in infile:
5   print line
6   lines.append(line)
7  infile.close()
8  print lines
9
10 mean = 0
11 for line in lines:
12  number = float(line)
13  mean = mean + number
14 mean = mean/len(lines)
15 print mean
```

Reading line-by-line (cont.)

Remark

Summing numbers is common, Python has `function sum` for the task

- `sum` operates on lists of floats, not on strings

Reading line-by-line (cont.)

Example

We use list comprehension to convert all elements in `lines` into floats

```
1 mean = sum([float(line) for line in lines])/len(lines)
```

An alternative implementation loads the lines into a list of floats directly

```
1 infile = open('data.txt', 'r')
2 numbers = [float(line) for line in infile.readlines()]
3 infile.close()
4
5 mean = sum(numbers)/len(numbers)
6 print mean
```

Alternative reading
Data from file

Alternative reading

It may seem confusing to see that a problem is often solved by many alternative sets of statements, this is in the nature of programming

Several solutions to a programming task must be judged and one that is either compact, easy to understand, and/or easy to extend is chosen

Alternative reading (cont.)

To deal with files, modern Python code uses the `with statement`

```
1 with open('data.txt', 'r') as infile:
2  for line in infile:
3   # process line
```

which is equivalent to writing the snippet

```
1 infile = open('data.txt', 'r')
2 for line in infile:
3  # process line
4 infile.close()
```

When using the `with statement`, there is no need to close the file

- Shorter code and better handling of errors if problems
- The syntax differs from the classical open-close pattern

Alternative reading (cont.)

Remark

Remembering to close a file is (was) key in programming

Alternative reading (cont.)

A `WHILE-loop` can read a file line by line using `infile.readline()`

```
1 while True:
2  line = infile.readline()
3  if not line:
4   break
5  # process line
```

The call `infile.readline()` returns a string containing text

- The text is the content of the current line in the file

`infile.readline()` returns an empty string at the end of file

Alternative reading (cont.)

The `WHILE-loop` runs forever, as the condition is always `True`

- Inside the loop, we test if `line` is `False`, and it is `False` when we reach the EOF, because `line` is an empty string
- When `line` is `False`, the `break statement` breaks the loop and the program flow goes to the statement after the `WHILE-block`

Alternative reading (cont.)

Example

Computing the average can now be done in yet another way

```
1  infile = open('data.txt', 'r')
2
3  mean = 0
4  n = 0
5
6  while True:
7   line = infile.readline()
8
9   if not line:
10   break
11
12  mean += float(line)
13  n += 1
14
15 mean = mean/float(n)
```

Alternative reading (cont.)

`infile.read()` reads a whole file, returns the text as `string object`

Example

An interactive session illustrates the use and result of `infile.read()`

```
>>> infile = open('data.txt', 'r')
>>> filestr = infile.read()
>>> filestr
    '21.8\n18.1\n19\n23\n26\n17.8\n'

>>> print filestr
    21.8
    18.1
    19
    23
    26
    17.8
```

- `filestr` dumps the string with newlines as `\n` characters
- `print filestr` is a prettyprint without quotes and with newlines

Alternative reading (cont.)

`String objects` have many useful functions for extracting information

`filestr.split()` splits the string into words (separated by blanks or a previously defined sequence of characters)

The 'words' in this file are the numbers in `data.txt`

Example

```
>>> words = filestr.split()
>>> words
    ['21.8', '18.1', '19', '23', '26', '17.8']

>>> numbers = [float(w) for w in words]
>>> mean = sum(numbers)/len(numbers)

>>> print mean
    20.95
```

Alternative reading (cont.)

More compactly, ...

Example

```
1 infile = open('data.txt', 'r')
2 numbers = [float(w) for w in infile.read().split()]
3 mean = sum(numbers)/len(numbers)
```

Reading text and numbers
Data from file

Reading text and numbers

Many data files contain a mixture of text and numbers

Example

The file `rainfall.dat` from **World Climate** (click me) is an example

```
1  Average rainfall (in mm) in Rome: 1188 months btw 1782 n 1970
2  Jan   81.2
3  Feb   63.2
4  Mar   70.3
5  Apr   55.7
6  May   53.0
7  Jun   36.4
8  Jul   17.5
9  Aug   27.5
10 Sep   60.9
11 Oct   117.7
12 Nov   111.0
13 Dec   97.9
14 Year  792.9
```

Reading text and numbers (cont.)

How can we read the rainfall data in this file and store the information in lists suitable for further analysis?

```
1 Average rainfall (in mm) in Rome: 1188 months btw 1782 n 1970
2 Jan   81.2
3 Feb   63.2
4 Mar   70.3
5 ...   ...
6 Dec   97.9
7 Year 792.9
```

A straightforward solution

1. Read the file, line by line
2. For each line, split the line into words
3. Store the first word (month) in one `list object`
4. Store second word (average rainfall) in another `list object`

Elements in the second `list objects` need be `float objects`

- If we want to compute with them

Reading text and numbers (cont.)

```
1  ##########################################################
2  def extract_data(filename):                              #
3                                                           #
4   infile = open(filename, ;'r')                           #
5   infile.readline()                       # skip the first line #
6   months = []                                             #
7   rainfall = []                                           #
8                                                           #
9   for line in infile:                                     #
10   words = line.split()                                   #
11   months.append(words[0])                  # words[0]: month #
12   rainfall.append(float(words[1]))       # words[1]: rainfall #
13  infile.close()                                          #
14                                                          #
15  months = months[:-1]                  # Drop the "Year" entry #
16  annual_avg = rainfall[-1]          # Store the annual average #
17  rainfall = rainfall[:-1] # Redefine to contain monthly data #
18                                                          #
19  return months, rainfall, annual_avg                     #
20 ##########################################################
21
22 months, values, avg = extract_data('rainfall.dat')
23
24 print 'The average rainfall for the months:'
25 for month, value in zip(months, values):
26  print month, value
27 print 'The average rainfall for the year:', avg
```

Reading text and numbers (cont.)

The first line in the file is a comment line, of no interest to us

- We read this line by `infile.readline()`
- We do not store the content in any object

The `FOR-loop` over lines in the file then starts from second line

Reading text and numbers (cont.)

```
1 Average rainfall (in mm) in Rome: 1188 months btw 1782 n 1970
2 Jan   81.2
3 Feb   63.2
4 Mar   70.3
5
6 ...   ...
7
8 Dec   97.9
9 Year 792.9
```

We store all data as 13 elements in the `months` and `rainfall` `lists`

Thereafter, we manipulate these `list objects` a bit

- `months` must contain the name of the 12 months only
- `rainfall` should contain the corresponding pluviosity

Annual average is removed from `rainfall`, stored in a separate variable

Reading text and numbers (cont.)

```
Average rainfall (in mm) in Rome: 1188 months btw 1782 n 1970
Jan  81.2
Feb  63.2
Mar  70.3

...  ...

Dec  97.9
Year 792.9
```

Remark

```
...

 months = months[:-1]                    # Drop the "Year" entry #
 annual_avg = rainfall[-1]             # Store the annual average #
 rainfall = rainfall[:-1] # Redefine to contain monthly data #

...
```

The `-1` index corresponds to the last element of a `list`, and the slice `:-1` picks out all elements from beginning to the last element, excluded

Reading text and numbers (cont.)

Shorter code, months and rainfall values are stored in a nested list

```
1  ##########################################################
2  def extract_data(filename):                              #
3                                                           #
4   infile = open(filename, 'r')                            #
5                                                           #
6   infile.readline()                      # skip the first line #
7                                                           #
8   data = [line.split() for line in infile]                #
9   annual_avg = data[-1][1]                                #
10  data = [(m, float(r)) for m, r in data[:-1]]            #
11                                                          #
12  infile.close()                                          #
13                                                          #
14  return data, annual_avg                                 #
15 ##########################################################
```

Data to file
User input, error handling and modules

Data to file

Writing data to file is comparably easier

There is one main `function` to pay attention to, `outfile.write(s)`

- It writes string `s` to a file handled by the `file object outfile`

Unlike `print`, `outfile.write(s)` does not append a newline character

- If string `s` is meant to appear on a single line in the file and `s` does not already contain a trailing newline character, it must be added
- It is often needed to add a newline character

```
1 outfile.write(s + '\n')
```

Data to file (cont.)

File writing is a matter of constructing strings containing the text we want to have in the file and for each such string call `outfile.write`

Data to file (cont.)

Remark

Writing to a file requires the `file object f` to be opened for writing

```
1 # write to new file, or overwrite file:
2 outfile = open(filename, 'w')
3
4 # append to the end of an existing file:
5 outfile = open(filename, 'a')
```

Writing a table
Data to file

Writing a table

Example

We want to write to file a `nested list` with tabular data

```
1 [[ 0.75,         0.29619813, -0.29619813, -0.75       ],
2  [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
3  [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
4  [-0.75,       -0.29619813,  0.29619813,  0.75       ]]
```

Writing a table (cont.)

```
1  data = [[ 0.75,         0.29619813, -0.29619813, -0.75       ],
2          [ 0.29619813,   0.11697778, -0.11697778, -0.29619813],
3          [-0.29619813, -0.11697778,   0.11697778,  0.29619813],
4          [-0.75,       -0.29619813,   0.29619813,  0.75       ]]
5
6  outfile = open('tmp_table.dat', 'w')
7
8  for row in data:
9   for column in row:
10   outfile.write('%14.8f' % column)
11  outfile.write('\n')
12 outfile.close()
```

We iterate through the rows (first index) in the list, for each row, we iterate thru column values (second index) and write them to file

- At the end of each row, we insert a newline character

Writing a table (cont.)

```
1  0.75000000  0.29619813 -0.29619813 -0.75000000
2  0.29619813  0.11697778 -0.11697778 -0.29619813
3 -0.29619813 -0.11697778  0.11697778  0.29619813
4 -0.75000000 -0.29619813  0.29619813  0.75000000
```

Writing a table (cont.)

To obtain this embellished result, add some statements to the program

```
1           column 1     column 2     column 3     column 4
2 row 1   0.75000000   0.29619813 -0.29619813 -0.75000000
3 row 2   0.29619813   0.11697778 -0.11697778 -0.29619813
4 row 3  -0.29619813  -0.11697778  0.11697778  0.29619813
5 row 4  -0.75000000  -0.29619813  0.29619813  0.75000000
```

For the column headings, the number of columns must be known

- This is also the length of the rows

```
1 ncolumns = len(data[0])
2 outfile.write(' ')
3
4 for i in range(1, ncolumns+1):
5  outfile.write('%10s ' % ('column %2d' % i))
6 outfile.write('\n')
```

Writing a table (cont.)

Nested `printf` construction: the text to be inserted is a `printf string`

We could also have written the text as `'column ' + str(i)`, then the length of the string would depend on the number of digits in `i`

Remark

The `printf` constructions is recommended for tabular output formats

- This gives automatic padding of blanks so that the width of the output strings remains the same

The tuning of the widths is commonly done by trial-and-error

Writing a table (cont.)

To add the row headings, we need a counter over the row numbers

```
1 row_counter = 1
2 for row in data:
3  outfile.write('row %2d' % row_counter)
4  for column in row:
5   outfile.write('%14.8f' % column)
6  outfile.write('\n')
7  row_counter += 1
```

Writing a table (cont.)

```
1  data = [[ 0.75,         0.29619813, -0.29619813, -0.75       ],
2          [ 0.29619813,   0.11697778, -0.11697778, -0.29619813],
3          [-0.29619813, -0.11697778,   0.11697778,  0.29619813],
4          [-0.75,       -0.29619813,   0.29619813,  0.75       ]]
5
6  outfile = open('tmp_table.dat', 'w')
7
8  ncolumns = len(data[0])
9  outfile.write('          ')
10 for i in range(1, ncolumns+1):
11     outfile.write('%10s    ' % ('column %2d' % i))
12 outfile.write('\n')
13
14 row_counter = 1
15 for row in data:
16     outfile.write('row %2d' % row_counter)
17     for column in row:
18         outfile.write('%14.8f' % column)
19     outfile.write('\n')
20     row_counter += 1
21
22 outfile.close()
```

Writing a table (cont.)

Iterate over the indexes of the `list`, as an alternative

```
1 for i in range(len(data)):
2  outfile.write('row %2d' % (i+1))
3  for j in range(len(data[i])):
4   outfile.write('%14.8f' % data[i][j])
5  outfile.write('\n')
```

Standard input and output as file objects

Data to file

Standard input and output as file objects

Reading input from keyboard was based on function `raw_input`

The keyboard is treated as a file

- It is referred to as **standard input**, `stdin`

The `print` command is used to print onto the terminal screen

The terminal is treated as a file

- It is referred to as **standard output**, `stdout`

Standard input and output as file objects (cont.)

Remark

All-purpose languages allow reading from `stdin` and writing to `stdout`

Reading and writing can be done with two types of tools

- `File-like objects`: `sys.stdin` and `sys.stdout`
- Special tools, `raw_input` and `print`

`File-like objects` behave as `file objects`, need no open/close

Standard input and output as file objects (cont.)

For input from keyboard, the two statements are equivalent

Example

```
1 s = raw_input('Give s:')
```

```
1 print 'Give s: ',
2 s = sys.stdin.readline()
```

The trailing comma in `print` statements avoids the newline added by default to the output string

Standard input and output as file objects (cont.)

Example

Similarly, the two statements are equivalent

```
1 s = eval(raw_input('Give s:'))
```

```
1 print 'Give s: ',
2 s = eval(sys.stdin.readline())
```

Standard input and output as file objects (cont.)

For output to the terminal window, the two statements are equivalent

Example

```
1 print s
```

```
1 sys.stdout.write(s + '\n')
```

Standard input and output as file objects (cont.)

It is handy to have access to `stdin` and `stdout` as `file objects`

Example

Suppose you have a function

- It reads data from a `file object`, `infile`
- It writes data to a `file object`, `outfile`

A sample `function`, `x2f`, may take the form

```
1 def x2f(infile, outfile, f):
2  for line in infile:
3   x = float(line)
4   y = f(x)
5   outfile.write('%g\n' % y)
```

Standard input and output as file objects (cont.)

This function works with all types of files `infile`, including web pages

- With `sys.stdin` as `infile` and/or `sys.stdout` as `outfile`, `function x2f` works with standard input and/or output
- Without `sys.stdin` and `sys.stdout`, code using `raw_input` and `print` would be needed to deal with standard input and output

Remark

This single function deals with all file media in a unified way

Standard input and output as file objects (cont.)

There is also something called **standard error**

- Usually, this is the terminal window
- As standard output

Programs can distinguish between writing ordinary output to standard output and error messages to standard error

In Python, standard error is a `file-like object`, `sys.stderr`

Standard input and output as file objects (cont.)

An application of `sys.stderr` is to report errors

Example

```
1 if x < 0:
2  sys.stderr.write('Illegal value of x'); sys.exit(1)
```

A message to `sys.stderr` is alternative to `print` or raising an exception

Standard input and output as file objects (cont.)

Redirecting standard input, output, and error

Remark

Standard output from some program `prog` can be redirected to a file output instead of screen, by using the 'greater than' sign ('>')

```
1 Terminal> prog > output
```

`prog` can be any program, also code run as `python myprog.py`

Remark

Similarly, output to the medium called standard error can be redirected

```
1 Terminal> prog &> output
```

Standard input and output as file objects (cont.)

Error messages are normally written to standard error

Example

- An example from a terminal session on a Unix machine

```
1 Terminal> ls bla-bla1 bla-bla2
2  ls: cannot access bla-bla1: No such file or directory
3  ls: cannot access bla-bla2: No such file or directory
4
5 Terminal> ls bla-bla1 bla-bla2 &> errors
6
7 Terminal> cat errors # print the file errors
8  ls: cannot access bla-bla1: No such file or directory
9  ls: cannot access bla-bla2: No such file or directory
```

Standard input and output as file objects (cont.)

Remark

Even when a program reads from standard input (keyboard), we can redirect the standard input to be from a file (say, with name `input`)

- Such that the program reads from this file instead of keyboard

```
1 Terminal> prog < input
```

Remark

Combinations are also possible

```
1 Terminal> prog < input > output
```

Standard input and output as file objects (cont.)

The redirection of standard output, input, and error does not work for Python programs executed with the `run` command inside IPython

- It only works with programs that are executed directly in the operating system in a terminal window, or with the same command prefixed with an exclamation mark (`!`) in the IPython

Standard input and output as file objects (cont.)

In a Python program, we can also let standard input, standard output, and standard error work with ordinary files instead

```
1 sys_stdout_orig = sys.stdout
2 sys.stdout      = open('output', 'w')
3
4 sys_stdin_orig  = sys.stdin
5 sys.stdin       = open('input', 'r')
```

- Any `print statement` will write to the output file
- Any `raw_input` call will read from the input file

Standard input and output as file objects (cont.)

Remark

Without storing the original `sys.stdout` and `sys.stdin objects` in new variables, these objects would get lost in the redefinition

We would not be able to reach the common standard input and output in the program